RL-TR-92-36
Final Technical Report
March 1992

DISTRIBUTED DATABASE INTEGRITY

SRI International

Ira B. Greenberg and Peter K. Rathman

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-92-36 has been reviewed and is approved for publication.

APPROVED:

Robert M. Flo

ROBERT M. FLO
Project Engineer

FOR THE COMMANDER:

John A. Granieri

JOHN A. GRANIERO
Chief Scientist for C3

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL(C3AB), Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1992	Final Oct 89 to Oct 90

4. TITLE AND SUBTITLE
DISTRIBUTED DATABASE INTEGRITY

5. FUNDING NUMBERS
C - F30602-89-C-0055
PE - 62702F
PR - 5581
TA - 21
WU - 94

6. AUTHOR(S)
Ira B. Greenberg, Peter K. Rathman

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SRI International
333 Ravenswood Avenue
Menlo Park CA 94025-3493

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3AB)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-92-36

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Robert M. Flo/C3AB/(315) 330-2805

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Distribution Database Integrity (DDBI) project is a research effort funded by Rome Laboratory and conducted at the Computer Science Laboratory of SRI International. The main goal of this effort was to improve data integrity and consistency management techniques for C^3 database application environments. The objective of this effort was to examine data integrity, consistency and concurrency control techniques for distributed object-oriented (O-O) database management systems. Specific areas that were investigated include: (1) rule-based concurrency control techniques for dynamic C^3 processing, (2) adaptive concurrency techniques for dealing with consistency versus availability of data, (3) temporal database mechanisms for retaining multiple versions of database objects, (4) rules, constraints, and application-specific knowledge-bases for integrity maintenance, (5) active triggers for automatically enforcing integrity rules based upon local or remote updates, and (6) the effect of incrementally mutable objects (IMO's) on distributed database integrity. The contribution is flexible, robust, data integrity techniques for distributed object-oriented database management systems.

14. SUBJECT TERMS
Distributed Systems, Database, Technology, Database Security, Integrity

15. NUMBER OF PAGES
72

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

Contents

Chapter 1

Introduction

The goal of the Distributed Database Integrity Project is to improve the integrity of distributed databases in a military command and control environment. An important aspect of this effort is to find ways to achieve an acceptable balance between the integrity of the database and its availability. The users' requirements for availability from the database system can become so critical that the users are willing to sacrifice some integrity in order to proceed. These problems are complicated by the fact that communication failures, including network partitions, can occur frequently in this environment.

During this project, we have examined a variety of approaches. In our interim technical report [GDM90], we describe techniques for adding temporal support [SA86] to the declarative constraint language of constraint equations [Mor89]. The resulting language can be used to improve integrity by writing dynamic constraints over historical versions of objects. It can also be used to improve availability by specifying transactions in a way that allows their inherent concurrency to be described and exploited. We also discuss techniques for combining different types of replica control algorithms.

This document describes a collection of approaches for improving the integrity and availability of the database by increasing the amount of information available to the database system and using knowledge-based techniques for reasoning about this information. We discuss the benefits of using an object-oriented data model and describe an approach called object contracts for managing object properties. Contracts allow structural and behavioral invariants about objects to be created and enforced. We discuss the use

of model-based reasoning to establish estimates and perform extrapolation. We present an approach called constraints with exceptions that allows processing to continue even though integrity constraints have been violated or have not yet been evaluated. We also discuss the inclusion of temporal support features in the object data model and the issues involved in object placement strategies. Finally, we present our conclusions and some ideas for future work. In another report for this project, the Feasibility Implementation Plan [GR90], we present a detailed description of the steps necessary to implement and evaluate these ideas, especially the ideas of object contracts and constraints with exceptions.

1.1 The Problem

Military command, control and communications and intelligence (C^3I) systems and the databases that support them must continue to function and provide reliable information, even in the face of extreme challenges.

One challenge to the data integrity, and hence to the reliability, of any information has its root in the intrinsic uncertainties of the C^3I task. Incoming information is often uncertain and contradictory, yet must still be sorted and evaluated. Data-entry errors and sensor failures may corrupt the data, and in a real-time system there will rarely be time for a manual audit process to uncover and correct such errors.

In addition to the uncertainty in the data itself, another threat to the integrity and availability of the database is from machine and communications failure. This threat is intermittent and caused by various kinds of crises. At times of crisis, especially if the system is under attack, the database must function and deliver good decision making information even when communications are unreliable and many sites are temporarily or permanently down.

We expect that most of the time, the system will not be actively under attack, most computers will be functioning, and communications will be reliable. However, just as a fire department buys equipment to be able to fight unusually large fires and an electric company must consider the power demand on the hottest summer day, the success of a C^3I system will be judged on how well it functions in a highly stressed situation.

This principle of designing for the worst case has profound implications for the architecture of a C^3I system and the database that supports it. If

we assume that periods of great stress are unusual, it is possible to greatly simplify both the system design, and this research effort. To be successful, any database must meet the needs of its users over a long period of time. This means that the database and its applications must be maintainable, and capable of evolving to meet changes in user needs. However, these capabilities, while important and the subject of much worthwhile ongoing research, are not the topic of this project. We assume that activities such as program debugging and schema evolution may safely be postponed until the end of a crisis. Instead, our research focuses on how a distributed object-oriented database (OODB) can best maintain availability and integrity while the system is under stress, and on how it can uphold its integrity in the face of uncertain and unreliable data.

1.2 Metaphors for a Highly Available Distributed OODB

No one has yet built an object-oriented database system (OODBS) with the extreme survivability needed for C^3I applications. Because we are envisioning a new kind of system, the question arises as to what the best operating metaphor is for such systems. Two main alternatives present themselves.

- A database supporting C^3I is best thought of as a database that happens to contain military information, rather than a military database per se. Such a database can be constructed using existing OODBs, which are themselves adapted from relational databases, by adding provisions for real-time function, enhanced survivability, and so on. In such a model, the demands of C^3I may cause some rethinking, but the system is fundamentally based on existing technology and ideas.

- Alternatively, we may decide that a C^3I system has needs fundamentally different from those met by any existing database, and a more radical redesign is needed. Because communication is problematic, the notion of a global state should be discarded, and the individual sites should be viewed as a collection of independent, but cooperating agents.

  The implementation of a system with no consistent global state would borrow many ideas from artificial intelligence, including:

- Cooperating agents, with explicit models of knowledge and belief
- Notification systems
- Model-based extrapolation systems

Of the above alternatives, this research project concentrates primarily on the first, viewing C^3I systems as extensions of OODBs. This is the more conservative approach, because it leverages an existing and well-developed technology. However, we will also consider the more radical alternatives, because they will provide indications for extensions and techniques to apply where conventional approaches fail.

1.3 Navigational Versus Value-Oriented Queries

An important difference between object-oriented and relational systems is the query model. Relational systems rely exclusively on value-oriented queries. In such queries, the system is given a logical specification of the desired results, and all tuples matching the specification are returned.

While OODBs also support some forms of value-oriented queries, a great proportion of object-oriented applications are primarily navigational. In navigational operation, an application traverses a hierarchy of subject and other specific links between objects. These links take the form of object ids stored with the referencing object.

This form of operation has a profound effect on the design of query processors, distribution, and integrity. First of all, query optimization by the database system is much less emphasized. The navigational application has a great deal of control as to the paths and orders by which data are accessed, and it is assumed that the programmer is able to make efficient choices. In addition, distribution semantics can be greatly different. For a value-oriented query, the query processor needs to know where the information is, and this requires access to a detailed fragmentation and distribution schema. While this schema does not record the location of each tuple individually, it does contain rules sufficient to assign any given tuple to a site. To keep such rules simple and easy to store, fragmentation is usually made along the lines of very simple predicates, so that, for example, different sites will store tuples with keys in different subranges of numerical values.

The distribution of objects does not have to be so carefully defined to support navigation in an OODB. If an application is searching for an object, and cannot find it in local storage, it knows that it must be at some other site. The application may have to use a registry service to find the exact location of an object, but the initial recognition that information is missing is intrinsic to the object structure, and is signalled by an id without a corresponding object. This means that object-oriented systems can afford to be far more dynamic and informal with regard to object distribution. A least-recently-used scheme may be employed to keep the most frequently accessed objects in local storage. More principled schemes, based on logical specifications and value subranges, are still possible, but they are not the only mechanism by which a site can determine its working set of objects.

Chapter 2

The ORION Database Model

Currently, there is no agreement as to the best model for object-oriented systems in general and this lack of agreement extends to the database domain. Several OODBSs have been built, each based on a somewhat different model. For concreteness, and because it is one of the best developed of the operational systems, we take as our starting point the ORION-2 distributed OODBS, which has been developed over the last five years at the Microelectronics and Computer Technology Consortium (MCC).

The current versions of ORION include objects with unique identifiers, attributes, classes, inheritance, methods, concurrency control, transactions, schema evolution, distribution, versions, and automatic translation between in-memory and on-disk representations for objects.[1]

Existing ORION implementations do not include a general trigger facility or a recovery mechanism for disk crashes. Neither of these features is inconsistent with the basic model, and each might be included in future versions.

Some of the advanced features of ORION are not directly relevant to this research project. As we argued in the introduction, schema evolution and long transactions will not take place when the system is under great stress. As long as these features are implemented so that they can safely be turned

[1] We list these features because the existence of a working ORION prototype provides an existence proof that this is a compatible set of features, all of which can coexist in the same system. Not all combinations of features are necessarily possible. For example, it has been argued [Ull88] that non-procedural, algebraic queries are fundamentally incompatible with a system providing object identifiers and methods.

off (i.e., aborted) as a crisis develops, we need not be concerned with them here.

Long transactions are motivated by a need to intermix "think time" with database updates. This happens, for example, in a design database where a designer may check out a portion of a design, work on modifying that portion, and upon completion try to integrate the modifications back into the existing shared database. The difficulty with this kind of modification is that if the designer locks the relevant section of the database upon checking it out, the database loses availability. Conversely, if no locking is used, some provision is needed for adjusting the new design to take into account any intervening updates. Therefore, some systems support change notification, and "soft" locks to facilitate the re-integration. However, in the midst of a crisis, there will rarely be enough time to perform this kind of update However, some kinds of network partitions may appear similar to long design transactions, so some of the techniques may apply.

Other features of the object-oriented data model might also cause problems for the integrity and availability of a distributed OODB. Unlike dynamic schema updates and long transactions, these features are more fundamental to the object-oriented model. These features include inheritance, change notification, and triggers. All can cause problems for availability because providing such features frequently requires access to data not contained in the object itself.

Inheritance, for example, implies that an object cannot be understood in isolation. Rather, the object's behavior and state are defined by both its own attributes and by any objects or class definitions from which it inherits. If an object is located on a different site than its ancestors, and communication with the ancestors' site is lost, then the behavior of the object becomes undefined or at least problematic. Because of this, the smallest manageable unit for data partitioning must include not just an object, but also any context required for the operation and interpretation of the object.

Change notification and triggers can cause similar problems, but for updates rather than for the interpretation of objects. In a system that supports triggers, what appears to be a simple update to a single object may in fact trigger a cascade of updates to triggered objects. Again, if these other objects are not available locally, completing the update is contingent on the availability of the communication system. These problems are not unique to the object-oriented model. One reason that inheritance and triggers are useful is

that they provide a way to model interconnections and dependencies present in the domain. Any knowledge representation that faithfully models such an interconnected domain will need to make use of some non-local features.

Chapter 3

Establishing and Managing Object Properties

3.1 Meta-Knowledge and Encapsulation

An idea central to the attraction of the object-oriented model is that objects are characterized by their behavior and object id. Any underlying structure used to implement the object behavior is encapsulated, and not available for direct manipulation. This encapsulation provides some of the most important benefits of the object model. For example, program reusability is greatly improved because programmers are free to vary the implementation of objects so long as the interface remains unchanged.

However, any intelligent system built on top of an object model will need to make use of meta-knowledge. One very simple kind of meta-knowledge is that required to maintain an index. Suppose we wish to be able to answer range queries about a set of objects very quickly. (An example of a range query is: "Find people with last names alphabetically between Go and Gr.") The natural thing to do in such a case is to create an index on the Last_name attribute, so that we may access the objects in sorted order.

In object-oriented systems, there is a fundamental difficulty with this approach. To maintain encapsulation, access to the Last_name attribute should only be via the object methods. The person's last name is a string that the object sends in reply to a given message, and the interface to the object defines no further operational semantics. In particular, there is no guarantee

that successive messages to the object will elicit the same response. (For example, suppose a hapless programmer tries to create an index on a set of file objects, and uses as the index attribute the get_next_char method!) Even worse, the method itself may be redefined. Finally, even if the method is normally well behaved, there needs to be a mechanism for the index maintainer to be notified when the attribute is updated.

The usual solution to this problem breaks the encapsulation of objects. The part of the system that maintains indexes is at a lower level of the system, and is empowered to look inside objects so that indexes may be maintained on structure rather than on behavior. In addition, all updates to the structure are passed through or at least made known to this index maintainer. In this way, when an attribute value changes, the maintainer can adjust the object's place in the index accordingly.

The disadvantage of breaking the encapsulation is that all code that maintains and directly manipulates such indexes is at a lower level than the object system. As such, the index maintaining code cannot be object oriented in itself, and it will not have the modularity and other benefits of the object model. In addition, by looking inside objects, the index maintainer is exposed to a great deal of complexity that is normally hidden by encapsulation. Maintaining an index on a polymorphic collection of objects is particularly difficult. These problems, which arise in the relatively simple task of index maintenance, are even worse for more involved applications.

In our system, there will be a great deal of meta-knowledge needed to enhance integrity and availability. In fact, it is likely that such information will form a substantial portion of the knowledge that the system manipulates. Therefore, it is imperative that the meta-knowledge be available at the ordinary programmer level and that tools of the highest quality are available for its maintenance. This implies an object-oriented paradigm. Therefore, we conclude that meta-knowledge should be handled so as not to break object encapsulation.

3.2 Object Contracts

We propose that each object be held responsible for maintaining its own meta-knowledge. Meta-knowledge is communicated to other portions of the system via a *contract*, which represents a promise on the part of an object

that some kind of invariant will be maintained.

For example, one kind of contract would be a declaration that a given method may be indexed. By signing this contract, the object is ensuring that calls to the method do not change the state of the object. In addition, the contract ensures that the other signatories to the contract will be notified whenever the attribute is updated, so that they may adjust their indexes accordingly.

Another example of meta-knowledge that can be declared in a contract is substructure independence. Under this contract, the object ensures that some portion b of its methods has behavior wholly independent from another portion a. Thus, it is possible to update portion b without worrying about portion a. Consequently, updates to substructures that have been declared to be independent need not be serializable. In its most grandiose form, each object would have a scheduler for intra-object concurrency control, which would take a stream of proposed queries and updates, and report any conflicts.

A contract that is nearly the converse is also useful in some situations. Such a contract would declare that groups of objects are linked, and that attempts to update one object in the group will necessitate, perhaps through triggers, updates to other objects in the group. These declarations would provide a warning to concurrency control algorithms that it may be necessary to lock the whole group. Between them, these two varieties of contracts provide granularity information, signaling the scheduler whenever the serialization granularity differs from the object granularity.

3.2.1 A Contract Protocol

Here we would like to give some examples of the kind of information that might be encoded in an object contract, and how it might be used. At this point it is premature to try to give a high level declarative language for defining contract terms. Instead, we will list example provisions, and describe how such provisions can be combined into contract agreements. We do not expect the list to be exhaustive, and indeed one of the requirements of the contract protocol will be an escape mechanism for adding extensions.

Perhaps the simplest kind of contract describes the basic query and update behavior of an object and its methods. The behavior of a method is a function of the internal structure of the object, but may also depend on other influences such as inheritance from other objects, data from I/O de-

Environment		Local State		
Writes	**Reads**	**Writes**	**Reads**	**Sample Task**
no	no	no	no	Cosine
no	no	no	yes	Access to a record attribute
no	no	yes	no	Blind write to a component
no	no	yes	yes	Incrementing a counter
no	yes	no	no	Access to system time service
no	yes	no	yes	—
no	yes	yes	no	—
no	yes	yes	yes	—
yes	no	no	no	—
yes	no	no	yes	—
yes	no	yes	no	—
yes	no	yes	yes	Reading a file via a local file pointer
yes	yes	no	no	Displaying a character on a screen
yes	yes	no	yes	—
yes	yes	yes	no	—
yes	yes	yes	yes	—

Table 3.1: Possible Values of the Gross Read/Write Contract.

vices, and so on. For convenience, we group all such influences together under the general heading of "environment." The result of a call to a method may depend on the internal structure of the object, the object's environment, or both. Similarly, a method may cause changes to either or both the internal structure and environment. This information can be captured in a contract. Table 3.1 lists the possible combinations and gives examples for some of the combinations. The information in such a specification is meant to be conservative so that a "no" provides a guarantee that the indicated read or write will not take place, while a "yes" is merely an indication that it is possible.

At the next level of complexity, we consider methods that are guaranteed to be pure reads and/or updates to local structures, and consider possible interactions between them. A likely question is whether a call to method A can affect the results method B will give in the future. In principle, any or all of these intermethod dependencies might be of interest, so if an object has n methods, one form of contract would be the $n \times n$ matrix of possible

dependencies. In practice, this might be far more information than is needed, so abbreviated forms may be used. Such forms might indicate that certain pairs of methods are independent or that all the methods in one subset are independent of all the methods in another subset.

For methods that do affect or read from the environment, a contract might still be used to codify any constraints on the scope of the effects. For example, a contract may guarantee that the effects of calling a method are limited to the objects in a given list. Similarly, a read method might list the ids of objects that may be consulted in answering a method call.

Yet another kind of contract would describe behavioral obligations. A typical behavioral contract would specify that whenever a specified attribute is updated, another object will be notified. This is very similar to the ideas underlying triggers. The difference is that a contract specifies that some behavior is promised, while a trigger is a piece of executable code that provides some behavior. Thus, a trigger may be used to fulfill the terms of a contact.

Let us now look at some possible applications and how they would use the above contract protocol.

- **Indexing.** To maintain an index on a class of objects, the index maintainer would need to know that the method is read-only, with no to all interactions except read from local structure. In addition, the object has a behavioral obligation to notify the maintainer in the event that the attribute is updated.

- **Update locking.** If the methods of an object can be partitioned into two independent sets, then the concurrency control manager need not enforce strict serializability at the object level. Instead, updates to each of the partitions can proceed independently.

- **Trigger Chasing.** Here the external behavior of an update is called into account. If an object can pre-specify which objects might be changed in an update, the updating transaction can attempt to prefetch those objects so as to achieve greater performance. In addition, when objects are distributed across sites in a network, an effort can be made to collocate objects with strong inter-dependencies.

- **Coordinating Future Behavior.** This is a much more advanced application, and assumes that the objects have agent-like characteristics.

The goal is for objects at different sites to coordinate behavior, even if future communication is likely to be cut off.

3.2.2 Sources of Knowledge for Contracts

Contracts are a special form of meta knowledge; as with all such knowledge-based systems, the problem of knowledge acquisition is paramount. For some contracts, knowledge acquisition will have to be the responsibility of the object programmer. For the most part, we would like to avoid imposing this burden. For the simpler kinds of contracts discussed above, such as (in)dependence of methods, and read/write of local storage/environment, much of the needed information can be obtained by analysis of the code that implements the methods. The techniques used would be very similar to the kind of analysis presently done in a globally optimizing compiler.

3.2.3 Inheritance and Object Contracts

We expect that the code that maintains contract invariants will be fairly complex. To avoid duplication and wasted effort, methods for determining and maintaining contracts will be inherited. In addition, contracts themselves, and their obligations will be inheritable. In the index example, a class definition may sign the contract that a particular method is to be indexable. All instances will inherit this obligation, as well as the list of beneficiaries who need to be notified in the event of an update. This inheritance of contract-maintaining functionality means that object contacts can subsume the functionality of more centralized techniques. For example, if we decide to write a centralized module that can look inside encapsulated objects and maintain indexes, we could simply package this module as an object and let all indexable objects inherit this capability.

3.2.4 Advantages of Object Contracts

The main advantage offered by the contract paradigm is flexibility. Meta-knowledge is available at the user level, and new forms of meta-knowledge may be defined as needed, rather than being "cast in stone" as low-level system services. Local autonomy is enhanced, because individual sites may

choose whether or not to participate in certain contracts, or may use different mechanisms to ensure compliance with the same contract. This decentralizes the effort of maintaining a given invariant, and may be of great advantage in a distributed environment.

For example, suppose that a given object is replicated, but takes on different forms at different sites: the object representation used at headquarters may be richer and more complex than is appropriate for efficiency and/or security reasons for field units. In the contract mechanism, there is no need for a single agent to understand the inner workings of each site's view of the object.

Chapter 4

Using Estimates and Extrapolations

C^3I systems function in a demanding real-time environment. A database serving such an environment must provide uninterrupted high-speed access to data. Unlike conventional databases, which take integrity to be an absolute and try to achieve the best possible availability within this constraint, C^3I systems will attempt to provide the best possible integrity within the constraint of absolute availability.

Suppose an application needs immediate access to a particular atomic data value, such as the amount of fuel remaining in a supply depot. Suppose also that the database is replicated so that a replica containing the needed value is immediately available. However, also suppose that not all replicas are kept completely up-to-date, so that the locally available value may not be the most current. In this circumstance, we propose a protocol where the result to a simple query is not a value, but a triple: ⟨Most-recent-value, Projected-current-value, Actual-current-value⟩. Most-recent-value is the value contained in the immediately accessible snapshot. Projected-current-value is a heuristically computed estimate of what the value is likely to be at present, while Actual-current-value is the globally consistent and up-to-date value. These three responses are delivered asynchronously so that the local snapshot is available immediately, the estimate is available after some local computation. and the globally consistent value may be arbitrarily delayed, perhaps in waiting for key sites to become available.

The success of such an architecture hinges on being able to compute good

estimates. Some methods are domain independent, and hence applicable to a wide variety of situations. In particular, statistical techniques can be used to extrapolate present and future values from past behavior. Such estimates can be treacherous, however, so it is best if they can be conditioned with domain-specific information such as constraints, domain models, and knowledge of future plans.

Statistical methods are most applicable to real-valued attributes. Where attributes have another form, we will have to rely almost entirely on domain-specific models and plans.

In general, the architecture of an estimation system can take advantage of both object orientation and the contract mechanism. Domain-independent methods, such as statistical estimation, can be stored in higher level objects, and the techniques inherited as needed. Domain-specific information will be stored so that its scope of inheritance will approximate its domain of applicability. In addition, if an application needs a response different than the usual three-level response, it can be arranged through the negotiation protocols of the contract mechanism. By pre-arranging a delivery format and semantics, the application can get the benefits of greater planning and optimization on the part of the object.

4.1 Model-Based Communication

Imagine a spectrum of communicating agents. On the extreme left side of the spectrum is a distributed relational database with global two-phase locking, voting algorithms, and so on. The algorithms for ensuring consistency are entirely syntactic, which is a great advantage because it means such databases will work no matter what the semantics of the application.

At the other extreme, consider a group of people, perhaps the circulating crowd at a high school reunion, pairs or small groups of whom meet from time to time and communicate. Each person's model of the world is unique: there is no sense of a globally consistent model. However, when two or more people meet, they try to synchronize, i.e., to bring each other up to date on important happenings. These communications are characterized by:

- A great deal of meta-communication ("have you heard about...?") aimed at determining what information should be transferred

- Models of other agents' thought processes

- The use of boredom and interest, as a rationing device to ensure that only the more important information is transferred in a world of limited time and bandwidth

- Very strong model-based compression, using the communications convention that objects or events are prototypical, and that interesting deviations from the prototype will be further explained

To apply some of these ideas to computer systems, we propose a predictor/corrector model of distributed systems. Each agent gains a description of the local situation by direct observation, and this local model is assumed to be accurate. In addition, each agent maintains an approximate model of the global situation. This model is based on whatever reports are available, filled in with global defaults and extrapolations.

In addition, each agent also maintains an approximate model of the local situation. This model is based on the history of reports given to outside agents, extrapolated forward to the current time. The purpose of this model is not to model reality, which is available by direct observation, but to model the state of mind of other agents. When the local model and the observed reality differ, this indicates that something unexpected has happened and that there is a high priority requirement to bring other agents up to date about this exceptional event. If there is uniformity of extrapolating techniques among agents, the communication can also be compressed. The messages need describe only the differences between expected and observed reality, with the assumption that anything not described is behaving according to expectations.

This is quite different from the syntactically based algorithms used in relational databases, because it hinges very heavily on semantics. Unless there are reasonably accurate prediction and extrapolation functions, the approximate model will differ so greatly from reality as to be useless. Thus, this kind of architecture will be useful only in domains in which large portions of the model are predictable. In addition to the domain, the time scale of operations will also affect the usefulness of these techniques. Weather, for example, is predictable at a fine-grained level over a period of a few days, but not over weeks or months.

Thus, this idea will need experimental investigation to determine how well it works.

Chapter 5

Using Constraints in an Object-Oriented System

A constraint equation is meant to provide consistency criteria among subsets of the data in a given extension of the database. The basic form of a constraint equation is:

Pattern op Pattern

where pattern is a type of query, and op is a logical test (such as subset inclusion) on the results of the queries given by the two patterns.

The patterns themselves are similar to relational algebra, structure and power.[1] The advantage of this formulation is that even a naive constraint verifier is not prohibitively expensive. Such a naive verifier would simply evaluate both of the queries and compare the results using the indicated test. The cost of verification is thus the cost of two relational queries.

Constraint equations are static conditions on a database state, and the code to maintain a constraint may be arbitrarily costly. For example, if we allow constraint violations to trigger actions, which may in turn cause other violations and trigger other actions, the resulting rule system is Turing complete. This allows great flexibility, because any needed computation may be encoded, but introduces a high level of complexity. In particular, as is shown by Theorem 1 in the appendix, the question of whether the constraint-action sequence ever terminates is undecidable. The proof of Theorem 1 relies

[1] One difference is that patterns can include an operator +, which allows a given edge to be traversed "one or more times," to provide a form of transitive closure.

on an unlimited ability to call rules to repair database constraint violations. This immediately suggests limitations that can make a constraint system more tractable and easily analyzed. For example, if all rules are guaranteed to terminate, and if only one rule or some bounded number of rules is called in response to a constraint violation, and if these rules cause no additional violations themselves, then we can be sure that the constraint maintenance process will terminate.

5.1 Specifying Transactions with Constraints

Database patterns can include a time specification, so that the pattern selects not just the value of a particular attribute, but the value of that attribute at a particular point in time. The notation is general, but by far the most common time values used are the before and after values for a given transaction. This allows constraint equations to express dynamic constraints on transactions, and in some cases actually to define transactions, in that the new database state is chosen so as to satisfy the constraint. In order for transactions to be fully defined by these constraints, a number of conventions must be added to the transaction language.

The first element needed is a concept of causality, or at least a separation between dependent and independent variables. In the absence of a causality specification, it would be possible to satisfy the constraint that expresses a transaction by picking an arbitrary new value, and changing the past to fit. Most of the time, this is not the desired semantics.

Even with a convention that the past determines the future and not vice versa, there remains a problem of ambiguity. For a given old state, there may be many new database states that satisfy the constraint equation. For example, database elements that are not mentioned in the transaction patterns could take on arbitrary values in the new database state. Thus, we need to take on a convention of minimal change semantics to eliminate this possibility. Also referred to as the frame assumption, minimal change semantics cause as few changes as possible to the database, consistent with the update requirement. However, even with causality and minimal change semantics, there may still be ambiguity about the best choice for the new

database state.

The advantage of using constraints to specify transactions is that we now have a more declarative representation, and that transactions may be more easily analyzed.

The program for concurrency control suggested by the interim report, but not carried out, is first to specify all transactions with constraint equations. Given the declarative representation, the read and write sets of the transactions can be analyzed. In fact, sometimes the analysis will reveal that a given transaction is composed of independent components, with disjoint read and write sets. If this is the case, we will be able to schedule the components independently, with increased concurrency.

Interestingly, this advantage is not unique to constraint equations. In fact, given any language in which it is possible to determine on which inputs a particular output functionally depends, it is sometimes possible to safely factor a transaction into separately schedulable subtransactions.[2]

The main use of constraint equations is to specify integrity constraints, and possibly triggers. There might be some benefit in having the integrity constraints and transactions represented in the same language. For example, we may be able to do some compile-time checking to determine that a transaction cannot violate a given constraint.

[2]Although, if we use strict two-phase locking and do not break the atomicity of the original transaction, at some point the parallelized aggregate transaction will need to hold all the locks that the original transaction needed at its peak. Thus, the opportunity to gain additional concurrency is limited to the advantage we might get from the parallelized transaction running faster.

Chapter 6

Constraints with Exceptions

There seem to be (at least) three distinct levels of database constraints, differing in their possible sources of violations, and also in the appropriate responses to them.

At *the first level*, *some constraints* are absolutes, intended to maintain algorithmic and representational invariants. Examples are:

- If an object is deleted, all references to it should be invalidated.
- The objects in a given B-tree are in sorted order.
- If B is a member of A's parent slot, then A is a member of B's child slot.
- If copies of an object are available at different sites across a network, all copies have identical values and behavior.

Statements in this class make no assumptions about the outside world. They are purely internal to the database and its underlying physical representation, and are for the most part artifacts of redundancies in internal data representations.

This type of constraint should not be violated at all, except perhaps temporarily, inside atomic transaction code. Other than this, any violation is indicative of a fault in the database system. Checking such constraints is very useful, as a part of "defensive programming." When a violation is found, great priority should be attached to its correction, because such violations

represent a threat to the integrity of the lowest levels of the database system, and can easily propagate to destroy large portions of the database.

At the next level, there are simple assumptions about fundamental properties of the world, that we do not expect to see violated. Examples include:

- Two physical objects may not be in the exact same position at the same time.
- Energy is conserved.
- Information does not travel faster than the speed of light.

These constraints do encode assumptions about the state of the world, and we may often expect the data in the database to indicate that these kind of constraints are violated. When a violation occurs, it may mean that we need to rethink assumptions or, far more likely, there is an error in that the database does not correctly reflect the true state of the world.

Finally, at the third level, there are constraints that represent policies, legal requirements, and observed behavior that happen "almost always." Examples include:

- Except in cases of extreme emergency, never draw the fuel supply at depot number 7 down below 20%.
- Grade 2 programmers are paid between $26,000 and $39,000 per year.

Here, many more explanations for a reported violation are possible. Just as for the lower levels, software bugs and mismatches between the data and the true state of the world can also cause violations at the third level. In addition, however, the possibility arises that the database does faithfully reflect the state of the world, but the world is not behaving according to expectations.

Now let us consider appropriate responses to detected violations of these constraints.

- If a failure at level 1 occurs, this means the data structures are inconsistent. Uncorrected errors of this sort are very dangerous, and responses up to and including halting the machine may be appropriate. In a distributed database, the possibility arises that different copies of

replicated data may become inconsistent. This is a level 1 constraint violation. Ordinarily, voting or primary token algorithms can rule out this possibility, but because of availability concerns these solutions are not desirable. Therefore, the database will need to recover gracefully even from some level-one constraint violations.

- Discovery of violations at level 2 indicates an error somewhere in the database. The proper correction of the error may be ambiguous, however, as the cause may be unclear. One suspect is the transaction that caused the violation. This transaction may be incorrect, in which case it should be aborted. However, it is possible that the transaction is correct, but is exposing a previously undetected error in the database. Therefore, a correcting response may have to be delayed until further investigation can determine the correct cause of the error.

- At level 3, many of the violations we see will occur because the world is not behaving according to plan, rather than because the database has incorrect data about the state of the world. If so, it may be appropriate to initiate corrective external action, for example to enforce the policy, but almost never should the database choose to "see no evil," and change its internal state.

Thus, especially for violations at levels two and three, the database must continue functioning in the presence of known constraint violations. This implies that constraints are not absolute and raises the interesting question of how to represent and manage non-absolute constraints and their exceptions.

6.1 Managing Exceptions

If a constraint might have some exceptions, the naive mapping from constraints to assertions in first order logic will not work. Under such a mapping, a violation gives rise to a database inconsistency, and under the normal semantics of first order logic, all inconsistent systems are essentially equivalent. Therefore, the formulation of constraints needs to be less absolute than the straightforward mapping.

One possible approach is given by Borgida in his work on exceptions in information systems [Bor85]. In this approach, constraints are reformulated

dynamically in the face of known exceptions. If Ψ is the formula for a constraint violated by database state db, and the exception can be blamed on the value of a predicate P on a vector of arguments β, a new constraint Ψ^* is constructed by the rule,

$$\Psi^* = \Psi \quad ,$$

with every occurrence of $P(\omega)$ replaced by

$$[(\omega \neq \beta) \wedge P(\omega)] \vee [(\omega = \beta) \wedge \neg P(\omega)] \quad .$$

The database will be consistent with this modified constraint, but this has the disadvantage that every database update is potentially a constraint update, and that the resulting constraints may grow excessively large and hard to verify.

Another approach is to weaken the constraint before any exceptions are noted, but to add a non-monotonic component that restores some of the strength of the original rule. For example, let us suppose that a certain agency has a rule that pilots can be no more than 6 feet tall. This might be expressed by a logical rule:

$$\forall x \mathit{pilot}(x) \Rightarrow \mathit{height}(x) \leq 6 \quad .$$

However, if there ever is a pilot who breaks this rule, it will cause a contradiction. Therefore, we might want to weaken the rule by allowing for the possibility of exceptions:

$$\forall x \mathit{pilot}(x) \Rightarrow [\mathit{height}(x) \leq 6] \vee \mathit{abnormal}(x) \quad .$$

However, without additional provisions, this weakening may make the database vacuous. If we use only the second rule, what is to rule out the possibility that all pilots are abnormal?

The usual approach here is to use a system with semantics such that pilots are assumed to be normal until there is evidence to the contrary. This kind of reasoning is formalized in artificial intelligence by such systems as circumscription, default logic, epsilon semantics, negation as failure, and many others. Collectively, such techniques are referred to as systems for non-monotonic reasoning.

Object-oriented systems already have features that embody some aspects of non-monotonic reasoning. One such aspect is the closed-world assumption,

the assumption that the set of objects represented in the database contains all the objects relevant to the domain at hand. For example, in a company it is usually convenient to make sure that all employees are represented by objects in the employee database. If the database is complete, it is possible to answer questions such as "How many employees are there?" with confidence. Note that this assumption corresponds to a possibly infinite set of facts of the form "X is not an employee." The assumption is non-monotonic, because the addition of a new fact, i.e., the discovery of a new employee, can lead to the invalidation of old conclusions. The closed-world assumption is not unique to object-oriented systems; much of the original work on this assumption was aimed at explaining the semantics of relational databases.

In our approach, we shall represent constraints with exceptions by using the closed-world assumption with some operational conventions, in order to achieve the desired non-monotonic behavior. We shall use the semantics of circumscription as a comparison.

We have a two-level procedure for dealing with exceptions to constraints. The transaction, if it detects violations, will flag the offending object(s). Later, an asynchronous cleanup procedure will try to do something intelligent about the violations.

6.2 A Proposed Architecture for Constraints with Exceptions

Suppose we have an integrity constraint expressed by the rule:

$$Rule_n : \forall xyz P(x,y,z) \quad ,$$

where $P(x,y,z)$ is an assertion over a database extension. If we wish to make $Rule_n$ allow exceptions, we could rewrite the rule to be

$$Rule_{n'} : \forall xyz P(x,y,z) \vee abnormal_n(x,y,z) \quad .$$

Note that we use a different flavor of abnormality for each rule, so that if a given object violates a constraint, we know which constraint it has violated. Note also that it is the entire rule-binding that is flagged as abnormal. Thus, if there is an exception to a rule that mentions several objects, all of the objects as a group will be abnormal.

6.3 Design of Constraints

A constraint exception handling capability makes the overall system more robust in the face of missing and incomplete information. Another advantage, which should not be overlooked, is the potential simplification to the constraint writing process. In a system where all constraints must be satisfied at the end of every transaction, an incorrect constraint can be an availability disaster. If a constraint is too tight, correct and perhaps essential transactions will not be allowed to proceed. On the other hand, a constraint that is too loose can allow incorrect transactions to execute, putting the database in a state that may disallow further transactions.

For example, suppose a given company has trouble setting an upper bound for salaries, and so checks salaries only to make sure that they comply with minimum wage laws. However, departments do have budgets, and there is a requirement that the total salary of all employees of a department is less than the department's salary budget. Suppose that the order is then given to raise a given employee's salary from $45,000 to $50,000. However, through a keypunch error the new salary is actually recorded as $500,000. This transaction is allowed to proceed, because there is no notion of a salary being "unreasonably large." However, this transaction uses up almost all of the money allotted for raises in a large department. Thereafter, many correct transactions will be disallowed on the basis of lack of funds. This might have been avoided if the system had a concept of "unreasonably large." If we allow the system to proceed with exceptions, constraint writers can take advantage of this by writing constraints not to a standard of "violations cannot happen" but to a weaker standard of "violations indicate probable errors."

This raises the question of tradeoffs between the strength of constraints and the severity of errors. We will use an example to explore how this tradeoff might be optimized. Suppose we wish to represent the constraint that employee ages should have "reasonable values," as a guard against data-entry errors.

Suppose the actual ages of employees are spread according to the distribution function $Age(x)$ in Figure 6.1, so that the proportion of employees with ages between n and $n + \delta$ is given by:

$$\int_n^{n+\delta} Age(x)dx$$

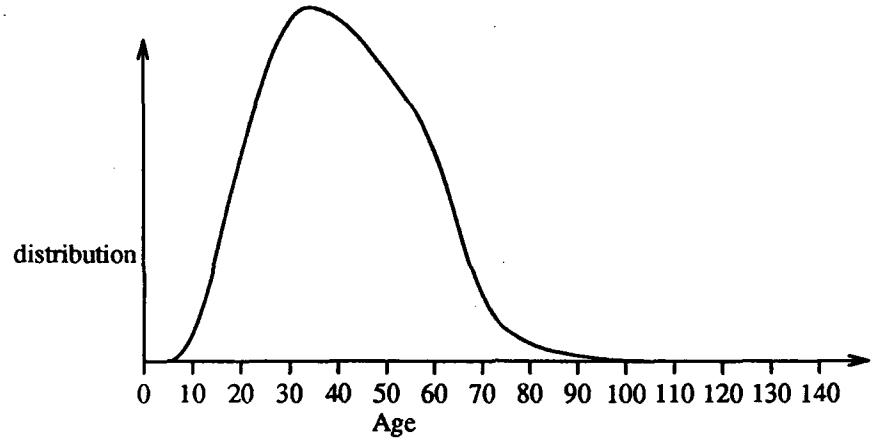

Figure 6.1: Distribution of actual ages

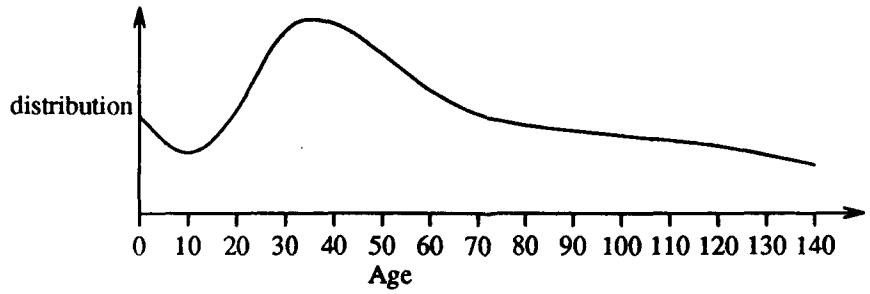

Figure 6.2: Distribution of error entries

However, assume that some proportion p of the age values entered into the database are in error for some reason, and these error values are distributed according to the function $ErrorAge(x)$ in Figure 6.2. Note that the error values show a much higher standard deviation than the correct values. Thus the observed distribution of input values is given by

$$observed(x) = (1 - p) * Age(x) + p * ErrorAge(x) \quad .$$

Then, for a given input data value, we can compute a retrospective probability of being in error p_e as

$$p_e(x) = \frac{p * ErrorAge(x)}{p * ErrorAge(x) + (1 - p) * Age(x)} \quad .$$

A graph of this probability would look something like Figure 6.3. As we see in Figure 6.3, as we reach more extreme ages, the probability of an input data value being in error rises to a near certainty. If we assign system costs $cost_u$ to uncorrected errors and $cost_f$ to false alarms, and if $p_e(x)$ follows a smooth bathtub-shaped curve as in Figure 6.3, then we should assign the upper and lower constraints at the boundaries where

$$p_e(x) * cost_u(x) = (1 - p_e(x)) * cost_f(x) \quad .$$

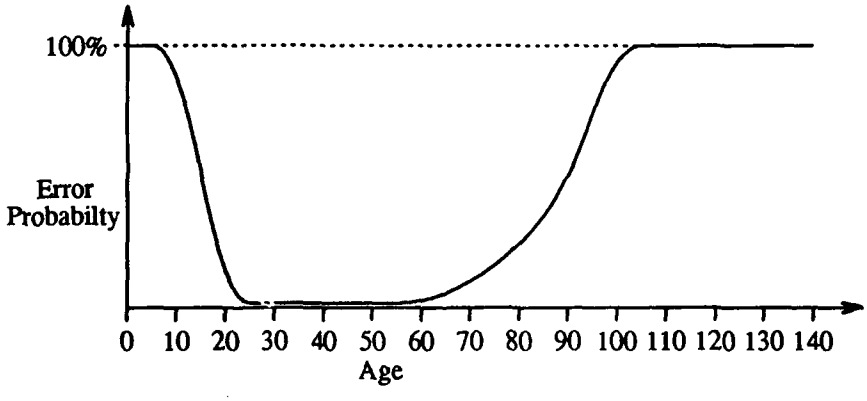

Figure 6.3: Probability that a given entry is in error

In practical situations, the above distributions are likely to be known only as approximations, and we may not have an exact measure of the costs of

uncorrected errors or false alarms. However, this exercise is meant to give a flavor of the added flexibility possible if constraints can have exceptions. Rather than requiring constraints to be rigorous absolutes, we assume a constraint becomes useful if adding it results in a net increase in utility for the database.

Chapter 7

Modeling Time

7.1 The Relational Approach to Time

Many of the key ideas regarding time-varying data and timestamps in general have been explored in the context of the relational model. We briefly review some of the relevant concepts here, and show how they apply to OODBs.

A relation is a set of identically structured tuples, each of which contains a ruling part (here shown underlined), and some number of dependent attributes:

⟨Key, attr1, ..., attrn⟩ .

If the data in a tuple are valid only at a particular time, or over a particular interval, we add time-value as an extra attribute to the relation:

⟨Key, attr1, ..., attrn, time-value⟩ .

Note that the time value is underlined. This is to indicate that some of the dependent attributes may depend on time: the key alone no longer determines them. Therefore, time-value is included in the ruling part.

However, some attributes may not change with time. Such stable attributes will only be a function of attributes of the original key. This dependency on a subset of the ruling part violates second normal form, and we can avoid redundancy and update anomalies by decomposing the relation into two separate relations:

⟨Key, time-value, time-varying-attributes⟩ .

⟨Key, stable-attributes⟩ .

So far, we have been deliberately vague about the definition of time-value. One of the most successful interpretations of time values is Wiederhold's interval representation [Wie90, KW90]. In this representation, the fundamental descriptor of time is the half-open interval. Several different kinds of semantics may be attached to these intervals, but the simplest is the *stability* assumption, in which if $[t_1, t_2)$ is the time value of a tuple T, then the data values of T are assumed to hold at all times starting with t_1, and up to but not including t_2. This means that a single tuple corresponds not to an atomic data value, but to a set of data values.[1] The semantics of relations also change. Usually, a relation is considered to be a set of tuples. But, in the interval representation, in which each tuple is a set, the more useful semantics for a temporal relation is that of a union of all the tuple sets. Because of this, syntactically different temporal relations may have the same semantics. For example, the relation:

Name	Salary	Time
John	12,000	[1960, 1961)
John	12,000	[1961, 1964)
John	14,000	[1970, 1969)

is equivalent to the single-tuple relation:

Name	Salary	Time
John	12,000	[1960, 1964)

The first two tuples coalesce into a single tuple over a longer interval, while the third tuple can be eliminated because its time interval is equivalent to an empty set. This raises the concept of *temporal normalization*: a temporal relation is normalized if all tuples with empty time intervals have been eliminated, and any pairs of tuples with adjacent time intervals and otherwise identical attributes have been coalesced.

In this representation, temporal versions of the relational operations such as selection, cross-product, difference, etc., can be defined. In each case,

[1] Using a tuple to represent a set of values is not unique to temporal databases. In geometric databases, n-dimensional intervals (i.e., a set of points) may be used to represent the extent of objects, while in databases that admit uncertainty, certain kinds of nulls represent a set of possible data values. In all these cases, special versions of the relational operators must be defined and implemented to take into account the modified semantics.

adjustments are needed to handle interval attributes. For example, if we applied the selection condition Time $\geq$ 1963 to the above single-tuple relation, we would get the result formed by determining what subset of the interval satisfies the selection condition:

Name	Salary	Time
John	12,000	[1963, 1964)

Similarly, to compute an equi-join along time-values, we form a cross product as usual, but instead of testing intervals for syntactic equality, we perform an intersection.

Relation A	
A-Value	Time-interval
a_1	[1,3)
a_2	[3,7)
a_3	[7,10)

Relation B	
B-Value	Time-interval
b_1	[1,4)
b_2	[4,8)
b_3	[8,10)

For example, suppose we have relations A and B and wish to perform the join $A \bowtie B$. This is equivalent to a selection on the cross product of A and B. This selection should be interpreted as acting on the sets of times denoted by the intervals, rather than on the intervals themselves. Thus, the correct semantics are obtained by replacing a pair of time intervals by the intersection of the intervals, and then normalizing the result.

$A \bowtie B$		
A-value	B-value	Time-interval
a_1	b_1	[1,3)
a_2	b_1	[3,4)
a_2	b_2	[4,7)
a_3	b_2	[7,8)
a_3	b_3	[8,10)

In the above example, the argument relations are in the from of *proper histories*: any given time value between the extreme start and finish points is contained in the interval of exactly one tuple for a given time-independent ruling part. For relations in this form, a temporal join may be computed with a sort-merge algorithm, at cost roughly comparable with that of a normal equi-join.

7.2 Time in the Object Model

Temporal ideas from the relational model generally translate to the object-oriented model, but additional issues need to be addressed because of representational differences. In the relational model, a time value of a tuple is assumed to apply to all of its attribute values. However, because the relational concepts of decomposition and normalization do not apply in the same way to objects, an object-wide scope for a time value may be inconveniently broad. Therefore it is sometimes necessary to define reduced scopes and corresponding storage architectures.

For example, suppose that among the slots of a given person object, there are two time-varying attributes, weight and salary. If both attributes are represented by proper histories over the same time domain, we can use the join techniques defined in the relational model. The whole-object history is computed as the join of the attribute histories, so it is possible to switch between putting timestamps on the individual attributes and on the whole object. We could, for example, store attribute histories to minimize storage redundancy, and still provide a method to access the whole-object history. When data are incomplete, or different semantics apply to intervals, the algebra is not quite as simple or clean. If this occurs, there might be information loss in converting back and forth between attribute-level and object-level time values. One representation is likely to be preferred, but the other can at least be approximated algorithmically.

In addition to questions of architecture, a number of semantic issues need to be examined, especially in a system that needs to maintain high availability despite poor communication. In Section 4, we saw how the database may need to substitute an extrapolation from an old value, if no fresh data is available. This presupposes that the system has a concept of time, including at a minimum, accurate timestamps and standards of how old data should be evolved. Object contracts also may involve time. In particular, when an object signs a contract, it may specify a limited duration for the applicability of the contract. In this way, if an object needs to change its behavior, the object has the option of waiting for the contract to expire, as opposed to the more laborious route of tracking down all the signatories and obtaining their permission to make a change.

Chapter 8

Patterns, Contracts, and Object Distribution

8.1 Distribution Costs

In this section, we draw together many of the ideas of this research project to examine the distribution design for an OODB.

Distribution design addresses the problem of where information should be located, and is motivated by the same considerations that lead to a distributed database. If data items can be located at or close to the sites where they are most used and updated, then the overall costs of distributed system operation will be minimized.

In most work on distributed databases, the measures of cost are response time and resource utilization. While these are important in our project as well, our primary focus is on availability and integrity. This shift in focus may not greatly change the preferred distributions, however. Communications lines are slow and costly, so communications should be minimized to reduce monetary costs. Communications lines are also unreliable, so to increase availability, as much information as possible should be stored locally.

8.2 Granularity of Distribution

For an OODB, the objects themselves provide an obvious granularity at which to consider the problem of data distribution. The database is a

collection of objects, and subsets of these objects will be allocated to the various sites. In many cases, however, this granularity is either too coarse or too fine for an optimal distribution. As we saw in the work on contracts, sometimes portions of an object can exhibit nearly independent behavior, and we may want to distribute parts of a single object to different sites. On the other hand, triggers and contracts can also bind sets of objects into such tight interdependencies that it is imperative to cluster such interdependent objects on the same site.

We need no new operations to consider clusters of objects. If, on the other hand, object-level granularity is too coarse, we need new operations for distributing parts of objects

8.3 Object Fragments

We shall postpone the question of when it is advisable to distribute portions of objects, and first introduce some operations to do so.

Database patterns provide a nonprocedural query language for accessing object databases; in addition, through the pattern's output form, they define the structure and content of the query result. The result of applying a database pattern to a particular object can be a surrogate that mimics some of the structure of the original object. We can even define methods for such a surrogate. For the most part, these methods will be simple accessors, but more complicated methods can be "inherited" from the original object. This surrogate can then be installed and queried on a different site than the original, and the pair together can provide a simulation of a distributed object. The actual procedure for accomplishing this would be something like the following.

1. Design a pattern to extract and structure a relevant subset of an object. The query result, an object in its own right, provides a type of materialized view of the original object.

2. Get the original object to sign a contract that it will notify the surrogate whenever the relevant portion of the original object is updated.

3. Install triggers in the update methods of the surrogate, so that any changes are propagated back to the original object.

4. Allow the surrogate to migrate to wherever it is most convenient. The surrogate will be able to field queries related to its scope, without requiring network access to the original object.

8.4 Distribution Design

The procedure described above gives us the ability to fragment and distribute objects. How do we find the wisdom to make best use of this ability? The answer lies in the various kinds of knowledge that can be available in an OODB. First of all, contracts provide a representation for a variety of meta-knowledge at the object level. Contracts that describe inter-method dependencies can be used to determine when one subset of the methods is nearly independent of other methods, and thus identify possible fragmentations for an object. At a larger scale, contracts can also indicate dependencies between objects, and thus provide guidance as to which objects should be collocated.

In addition, we need application-specific knowledge, in the form of execution profiles. Such profiles provide a statistical forecast of which applications are likely to be run at which sites, and which objects and methods are called by given applications. Acquiring such knowledge can be arbitrarily complex, but can make use of deep insight into the semantics of the applications. However, the simple empirical approach of running the database under a standard mix of applications and gathering statistics from such trials has had considerable success in the commercial world. In C^3I applications, the environment is inherently unpredictable, so far more care has to be taken to ensure that the distribution is robust across a range of configurations and possible faults. Nevertheless, at least a hybrid empirical method is likely to be the most successful..

Once the database interdependencies, applications, communications links, and possible faults are all modeled with reasonable accuracy, choosing a distribution design becomes an optimization problem to be addressed with heuristic or standard mathematical techniques.

Chapter 9

Future Work

This project has proposed several promising approaches for improving integrity and availability in military command and control databases. This chapter describes some work that could be done in the future to help realized the benefits of these approaches. The purpose of this work would be to design mechanisms to implement these approaches and to develop experimental prototypes to evaluate them. While these topics will be discussed individually, it will be important to design them so they will work together. Some of these topics, especially object contracts and constraints with exceptions have been discussed in detail in a companion document, the Feasibility Implementation Plan [GR90].

9.1 Object Contracts

The purpose of object contracts is to establish agreements among database objects and portions of the database system about object properties. Many details need to be worked out in order to develop efficient implementations and evaluate the benefits of this approach. Representations will have to be defined for the contracts themselves, and it will be necessary to determine what contract terms are suitable and how they can be represented. Negotiation protocols will need to be developed to allow contracts to be established, and then possibly changed to accommodate new information or requirements. A notification mechanism should also be designed; notification is an important kind of contract term and will help support parts of the

contract mechanism such as contract establishment and termination. These mechanisms and policies become more complicated when objects are replicated or fragmented. The contract terms and guarantees will have to be enforced by the database system in order for them to be meaningful.

9.2 Constraints with Exceptions

Constraints with exceptions allow constraints to be violated in a controlled manner. This approach also needs to be worked out in greater detail. First, the representation of constraint exceptions needs to be defined. Then, mechanisms and policies need to be developed to detect exceptions, to determine when and how to evaluate exceptions, and to remove exceptions. It will also be important to determine what it means to execute when exceptions exist, and how to interpret results based on abnormal values. One approach may be to employ compensation to overcome the effects of exceptions that turn out to be errors. The database interface will need to be enhanced so that users can review and evaluate exceptions and can observe their effect on results.

9.3 Model-Based Communication

To provide model-based communication, each agent, such as a database system, employs a model of some activity to organize, understand, and make use of the information it knows about that activity. Applying the model to the information it receives from other sites, the agent forms a view of the global state. Applying the model to the information it sends to other sites, it forms a view of how other agents see its current state. This approach could help improve availability and integrity in systems with slow communication and network partitions.

Because this approach relies heavily on the use of semantics, it will be difficult to apply it to all situations. Common situations that are amenable to modeling will have to be identified. Models will have to be developed for these situations, along with the associated functions such as those for prediction and extrapolation. It will also be necessary to work out representations for the models and their world views, and policies for when to use model-based

views and extrapolations. Methods based on models and world views could also be developed to manage the collection and dispersal of information in the distributed database system. In addition, the database interface could be extended to allow the user to see the influence of the models on the results.

9.4 Constraint Systems

Declarative constraint systems allow the relationships among data objects to be expressed, evaluated, and enforced. While they have been studied for a while, it has usually not been in the context of systems with slow communication and network partitions. A constraint system is an important base technology for constraints with exceptions and can be used in a variety of ways to improve integrity. A prototype containing a subset of current constraint ideas, such as constraint equations, could be specified and implemented to evaluate its benefits and costs, both individually and in conjunction with other proposed mechanisms. This subset and the related mechanisms and policies should be selected so that they can be implemented efficiently and constraint evaluation and enforcement can be guaranteed to terminate.

9.5 Active Databases

Active databases are able to take actions beyond those explicitly requested by the database users. Mechanisms for active databases are an important base technology for many of the ideas proposed in this report. They have not been studied extensively in the context of OODBSs. A prototype containing active database mechanisms could be implemented to support activities such as constraint evaluation and enforcement, notification, exception evaluation, and the use of compensating actions. The mechanisms for the prototype should be designed so that they can be implemented efficiently and guaranteed to terminate. An important question to be explored is whether to use low level mechanisms or higher level mechanisms with more advanced abstractions for specific tasks.

9.6 Intelligent Databases

Intelligent databases are able to form inferences about information available to them. This ability would be an important base technology for supporting ideas contained in this report, such as approximating the value of unavailable objects from models or contract guarantees. This technology is also relatively new and has not been well integrated with OODBSs. A prototype could be developed to explore the integration of intelligent database mechanisms with the other ideas that have been proposed. One possibility would be to select ideas from the persistent knowledge-base/database project that was recently completed at SRI [Hsi90].

9.7 Meta-Knowledge Interface

A typical database interface accepts query and update commands from users and returns data values and error responses. In this report, we have described how an enhanced interface that contained meta-information about the input and results could be used to improve integrity and availability. For example, the user could be allowed to specify temporal contexts for queries, the urgency with which results are needed, acceptable confidence levels for results, execution priorities, intended usage profiles, and so forth. The database could be allowed to respond with annotations indicating the source, recency, and accuracy of the results; whether the results are based on exceptions or estimates; multiple values indicating different versions of a result obtained from different means; and so on. A prototype for a meta-knowledge interface could be implemented to explore these ideas. It would be necessary to find ways to specify this information and to understand its meaning.

9.8 Temporal Support

The use of time has been mentioned in many ways on this project. For example, users could specify that they want results from a certain time interval, before a given event, or of a certain recency; constraints could be enhanced with time information so they can be used to specify transactions; historical

values could be stored so they can be explicitly requested, or can be used to detect trends and create estimates; results can be annotated with time information so that their value can be better assessed by users, and so on. A prototype could be developed based on an advanced temporal approach such as Wiederhold's time intervals [Wie90]. The temporal information would have to be integrated uniformly with all of the other mechanisms, and techniques for maintaining an historical OODBS would have to be studied.

Chapter 10

Conclusions

This report discusses a variety of approaches for improving the integrity and availability of military C^3I database systems. The integrity of the data is a key concern because of the critical nature of the data's uses. At the same time, users often require the database to be highly available so they can proceed in a timely manner. The urgency with which users need to operate can cause the availability of the database to become more important than the need for globally consistent data. The goal of the database system is to maintain as much integrity as possible while providing the required level of availability, and to restore integrity as soon as possible after it has been damaged. Network partitions and slow communication are difficult challenges facing integrity and availability.

A central theme underlying our approaches is to increase the amount of information available to and supplied by the database system. This meta-knowledge will allow the database system to act more intelligently with respect to its users' requirements, its contents, and its environment, and allow users to act more intelligently with respect to database responses. It enables the database system to distinguish among different situations and to adapt to changes, instead of treating all users, data, workloads, and configurations in the same way. For example, availability will be improved if the database system can determine that a user will be satisfied with globally consistent data that is up to a day old, instead of assuming that it must supply the most recent values. Active database and knowledge-based techniques are used to specify, manage, and reason about the meta-knowledge.

In chapter 2, we discuss the effect of using an object-oriented data model

on integrity and availability. In particular, we examined the data model supported by the ORION-2 database system, which we believe to be one of the best examples available. We conclude that object-oriented data models aid integrity by providing support for describing and enforcing structural relationships and behavioral information that is not present in current data models such as the relational and hierarchical models. Examples of the additional structural relationships include the IS-A, part-of, and composed-of relationships, and examples of the support for behavioral information include methods, inheritance, and encapsulation.

We also conclude that the object-oriented data model will have a mixed effect on availability. The additional links among database components means that more components of the database system may have to be available to perform operations. For example, performing an operation on an object may require access to methods inherited from the object's ancestors. However, this additional meta-knowledge can help guide replication and distribution decisions. Availability could be improved by trying to collocate information that will frequently be referenced simultaneously. The fact that links between objects are represented as explicit references to object ids can also improve availability by helping the database system realize when all relevant information has been located.

Chapter 3 presents a new approach for establishing and managing object properties called object contracts. An important problem with object-oriented data models is that strict encapsulation can cause inefficient performance because properties of the implementation cannot be exploited. Instead of breaking encapsulation, with all of the problems that can cause, object contracts allow relevant high-level implementation properties to be specified, and make it possible for those properties to be guaranteed for a certain time period. In this way, details about implementations are not scattered throughout the system and implementations can change at any time as long as they maintain the properties specified in contracts. New implementations can also change the properties specified in contracts, but only after the contracts have expired. The idea of contracts can be extended to cover the behavior of objects, such as agreements to notify other objects when events occur, and invariants on the contents of objects.

We believe that contracts will help improve both integrity and availability. Knowing information about an object's implementation will allow for better performance, which will in turn improve availability. For example, knowing

that two of an object's methods are independent will enable the database system to execute them in parallel and in a way that is not serializable. This information will also help the database system improve availability by making better object replication, fragmentation, and distribution decisions. Having contracts about object behavior and contents can help improve integrity and availability, especially during network partitions, by making it possible to reason about this information. As a result, the database system can respond more quickly, can provide answers that are more accurate, and can provide better assessments about the accuracy of answers.

In chapter 4, we present the idea of responding to a query with three different answers. The goal is to provide users with a range of choices for different levels of integrity and availability so they can make an appropriate tradeoff. One response would arrive quickly and would be based on information available locally. Another, presumably more accurate response, would arrive later because of the need to do some computation, and would be based on local information and models of remote or global information. The last response would be most accurate, but would not be available until all relevant information had been accessed. The model-based knowledge could also be used to help determine the most critical information to send to other sites in order to improve global consistency, and to allow sites to communicate fully with a smaller amount of information. We believe that this approach is promising, but the success of model-based estimation depends on the difficult problem of acquiring accurate models.

In chapters 5 and 6, we discuss issues related to the use of constraints. First, we describe how database systems that automatically take corrective actions to maintain constraints can become complex and difficult to analyze if limits are not imposed on them. This can lead to constraint maintenance activity that is unexpectedly expensive and may not even terminate. In the appendix, we prove that the question of whether the constraint-action sequence terminates is undecidable. We then consider some ramifications of using constraints to specify transactions, an idea that was presented in our interim report [GDM90]. We conclude that several conventions must be added to the transaction language to make this viable. In particular, the concept of causality is needed to prevent transactions from changing the past, and the convention of minimal change semantics is needed to prevent data items not mentioned in a transaction from changing.

Chapter 6 proposes a new approach for managing constraints, called con-

straints with exceptions, in order to improve the integrity and availability of the database. The basic idea is to make it possible to violate constraints in a controlled manner. Instead of rejecting violations or changing constraints to accommodate them, we propose to annotate the abnormal values. Additional support would be needed to detect, store, represent, resolve, and process with constraint exceptions. Availability can be improved by allowing processing to continue in a controlled manner despite values that violate constraints, values whose status has not yet been evaluated, and constraints that might be incorrect. This is especially helpful during network partitions and when processing is urgent. Integrity can be improved by writing tighter constraints. We describe how constraints with exceptions can be used to determine a lower average cost for an error by making it possible to establish a more advantageous tradeoff between false positives and false negatives.

In chapter 7, we present an interval representation of time values that has been proposed by Wiederhold for the relational data model. Much of this approach also applies to the object-oriented data model. One important difference is that, unlike the relational data model where a time value is assigned to a tuple and assumed by all of its attributes, an object-wide time value may sometimes be inappropriate. Instead, individual time values should be assigned to each time-varying object attribute. A whole-object history can then be computed from its attribute histories. This approach can help save storage, but its semantics need to be explored further. Temporal support will help improve both integrity and availability, and is assumed by other features discussed in this report.

Finally, chapter 8 discusses how some of the ideas proposed for this project can be used to improve object replication, fragmentation, and distribution decisions. Good decisions will help improve availability by locating information near where it will be needed and collocating information that will be referenced together. We propose an approach called surrogates for supporting object fragmentation, where a surrogate is similar to a view of an object. The location decisions can be guided by the knowledge represented in the object-oriented data model, constraints, contracts, and user profiles.

This report proposes a variety of approaches for improving the integrity and availability of a database system in a military command and control environment. They are based on increasing the amount of information available to the database system, and exploiting this information with knowledge-based techniques. Together, they will allow each user to receive an individ-

ualized tradeoff between integrity and availability, and allow a high level of integrity to be maintained while availability is increased.

Bibliography

[BMO+89] Robert Bretl, David Maier, Allen Otis, Jason Penny, Bruce Schuchardt, Jacob Stein, E. Harold Williams, and Monty Williams. The Gemstone Data Management System. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 12, pages 283–308. ACM Press, 1989.

[Bor85] Alexander Borgida. Language Features for Flexible Handling of Exceptions. *ACM Transactions on Database Systems*, 10(4):565–603, December 1985.

[GDM90] Ira B. Greenberg, Alan R. Downing, and Matthew Morgenstern. Distributed Database Integrity. Interim report, SRI International, Menlo Park, CA, April 1990. For RADC Contract No. F30602-89-C-0055.

[GR90] Ira B. Greenberg and Peter K. Rathmann. Feasibility Implementation Plan. Interim report, SRI International, Menlo Park, CA, November 1990. For RADC Contract No. F30602-89-C-0055.

[Hsi90] Donovan Hsieh. A Logic to Unify Semantic Network Knowledge Systems with Object-Oriented Database Models. CSL Technical Report SRI-CSL-90-15, SRI International, Menlo Park, CA, December 1990.

[KL89] Won Kim and Frederick H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications*. ACM Press, Addison-Wesley Publishing Company, 1989.

[KW90] Samuel N. Kamens and Gio Wiederhold. An Implementation of Temporal Queries for SQL. Submitted for publication, August 1990.

[Mor89] Matthew Morgenstern. Intelligent Database Systems. Final report, SRI International, Menlo Park, CA, May 1989. For RADC Contract No. F30602-86-C-0088.

[SA86] Richard Snodgrass and Ilsoo Ahn. Temporal Databases. *IEEE Computer*, 19(9):35–42, September 1986.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1988.

[Wie90] Gio Wiederhold. *Semantic Database Design*, chapter 7. McGraw-Hill, 1990. Chapter preprint from the forthcoming book.

Appendix A

Proof that Termination is Undecidable

Theorem 1 *A constraint system that allows cascading actions in response to constraint violations is Turing complete. Hence, questions of termination of such a system are undecidable.*

Proof: The proof of this theorem shall be constructive, i.e., we shall illustrate a database and constraint system that can simulate an arbitrary Turing machine computation. The database for the simulation consists of three relations.

FSA				
State	Read-Symbol	Write-Symbol	Move	Newstate

Current-State	
State	Head-Position

Tape	
Tape-Position	Symbol

One relation, FSA, gives the transition table of a finite state automaton. Another relation, Current-State, acts as a program counter. Its single tuple has attributes for the current state of the Turing machine and the position of the head on the tape. The third relation, Tape, is a sequence of symbols, indexed by the attribute Tape-Position.

There is only one constraint in the database, and it states that

Current-State.State == "Terminal,"

which is meant to indicate that the Turing machine program has successfully terminated. When this constraint is violated, an action code is called that attempts to repair the database to make the constraint true. The effect of calling this action code is to advance the Turing machine computation by one step. This action code may be encoded as a database pattern as shown in graphical form in Figure A.1. This pattern finds the tuple in relation FSA with State corresponding to Current-State.State, and Tape.Symbol corresponding to that currently under the tape head in relation Tape. Note that in the figure, certain attributes are marked with a "*" to indicate that these attributes are in the outform of the pattern. In this case, these attributes are used for assignments to the database.

```
Tape.Symbol* ← FSA.Write-Symbol*
Current-State.State ← FSA.Newstate*
Current-State.Head-Position ←  if FSA.Move* == "Forward" then
                                   increment(Current-State.Head-Position)
                               else
                                   decrement(Current-State.Head-Position)
```

We have chosen to express the assignments directly, although they can also be encoded in the pattern notation as constraints on a post-assignment version of the database. Executing the assignments moves the computation one step forward. If the new value written into Current-State.State is "Terminal," then the constraint is satisfied, and computation has terminated. If not, then the constraint is not satisfied, and the action rule is called again.

Because neither the relations FSA or Tape were specified, this constraint can be used to simulate any Turing machine on any input tape. Because the termination of a Turing machine is undecidable, so also is the termination of the constraint/action set. □

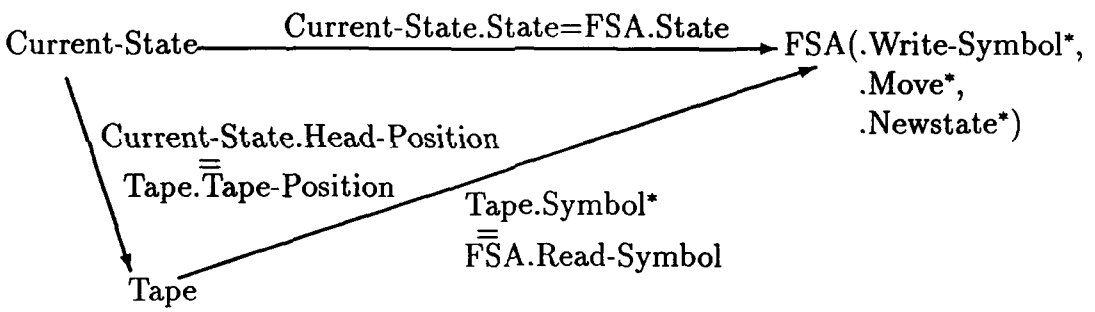

Figure A.1: Database pattern

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.